

04 | 跨越现实的障碍（上）：要性能还是要模型？

2021-07-01 徐昊

《如何落地业务建模》

课程介绍 >



讲述：徐昊

时长 24:17 大小 22.25M



你好，我是徐昊。今天我们来聊聊通过关联对象（Association Object）建模聚合（Aggregation）。

在前面三节课，我们学习了领域驱动设计中的“两关联一循环”：模型与软件实现关联；统一语言与模型关联；提炼知识的循环。其中，统一语言与提炼知识的循环作为一种更为平衡的权责关系，促进了业务方与技术方更好的协作。而这一切又是以模型与软件实现关联为基础。

然而落地到实践中，关联模型与软件实现总有一些让人纠结与苦恼的地方。引起这些问题的主要原因是**架构风格的变化**。我们已经从单机时代过渡到了多层单体架构，以及云原生分布式架构，但我们所采用的建模思路与编程风格并没有彻底跟上时代的步伐，这种差异通常会以**性能问题**或是**代码坏味道**的形式出现。

如果我们想要真正发挥出领域驱动设计的作用，就需要**在不同架构风格下，找到能够维持模型与软件实现统一的办法**。这也是这个领域常看常新，总能产生新实践的原因。

因而接下来，我会用三节课来介绍一组实现模式，帮助我们应对从单机架构过渡到多层架构，保持模型与软件实现的关联。这些模式也是我们后面学习在微服务和云原生时代，实施领域驱动设计方法的基础。

今天这节课，我们就先从关联对象这一方法开始讲起。关联对象是一个古老的设计 / 分析模式，Martin Fowler 在 [🔗 《分析模式》](#) 中讨论过它。Peter Coad 将它视为一种常见的业务构成模式，并应用到业务分析中。而我大概从 2005 年开始，使用它建模领域驱动设计中的聚合与关联关系，以解决领域模型（Domain Model）中对技术组件的封装问题。

不过，在讲解到关联对象的具体做法之前，我们需要先看一下**领域驱动设计中的聚合关系在具体实现中存在哪些问题**。了解了问题所在，我们才能更有针对性地去寻找解决办法。

无法封装的数据库开销

在应用领域驱动设计的时候，聚合与聚合根（Aggregation Root）是构成“富含知识的模型（Knowledge Rich Model）”的关键。通过聚合关系，我们可以将**被聚合对象的集合逻辑放置于聚合 / 聚合根里，而不是散落在外，或是放在其他无关的服务中**。这么做可以使得逻辑富集于模型中，避免“逻辑泄露”。

不过落到具体实现上，我们经常会遇到这样一个挑战，即：**这些被聚合的对象，通常都是被数据库持久化（Persistent）的集合（Collection）**。也就是说，数据库系统的引入，网络 I/O 与其他性能开销无法被接口抽象隔离。而将具体技术实现引入领域模型，则有悖领域驱动设计的理念。

接下来，我们就通过一个例子，在具体的上下文中看一看存在什么问题。然后，再看一下这个问题在建模思路上的根源，以及要如何修正我们的建模思路。

集合遍历与 N+1

让我们再回到在 [🔗 第二讲](#) 里展示的极客时间专栏的例子：

```
1 class User {
2
3
4
5     private List<Subscription> subscriptions;
6
7
8     // 获取用户订阅的所有专栏
9     public List<Subscription> getSubscriptions() {
10         ...
11     }
12
13     // 计算所订阅的专栏的总价
14     public double getTotalSubscriptionFee() {
15         ...
16     }
17 }
18
19
20 class UserRepository {
21     ...
22     public User findById(long id) {
23         ...
24     }
25 }
```

按照面向对象和领域驱动设计提倡的做法，User 作为聚合根，需要管控其对应的 Subscription。现在有一个简单且常见的需求：在页面上，显示对用户已经订阅过的专栏进行分页处理。那么，我们大概会这么写：

```
1 user.getSubscriptions().subList(from, from + pageSize);
```

[复制代码](#)

可以发现，在这段代码里，我们假设所有的 Subscription 都在内存里，因此所有的计算与逻辑都可以在内存中完成了。那么，我们就需要在读取 User 的时候，将它对应的所有订阅过的专栏信息都从数据库读取到内存中，以便后续使用。

如果是少量订阅的场景，比如几个或者几十个，那这段代码并没有什么问题。但如果有个特别好学的人，买了几万甚至几百万个专栏呢（先假设专栏有无限个）？此时将所有订阅过的专栏都读取到内存里，这就意味着会有巨大的网络 I/O 开销和内存占用。

当然，这里你可能会说，JPA/Hibernate 等 ORM 提供了延迟加载啊。是的，但这又会引入经典的性能瓶颈 **N + 1 问题**。因为随着延迟加载集合的遍历，其中的 Subscription 对

象会被依次加载。

延迟加载的实现流程是这样的：

先执行一条查询获取集合的概况。比如总共有多少条记录之类的信息。

然后根据概况信息，生成一个集合对象。这时候集合对象基本上是空的，不占用什么内存空间。

随后，当我们需要集合内的具体信息的时候，它再根据我们需要访问的对象，按需从数据库中读取。

理论上讲，这是为了避免一次性读入大量数据带来的性能问题，而提出的解决办法。

然而，如果需要获取所有的数据，那么我们总共就会有 $N+1$ 次数据库访问：1 次是指第一次获取概况的访问， N 次指而后集合中 N 个对象每个一次。而每一次加载，都伴随着对数据库的访问，自然就会带来 I/O 与数据库的开销。特别是频繁地对数据库访问，可能会阻塞其他人，从而造成性能瓶颈。

在这种情况下，我们其实没有什么好的选择：**要么是一次性读入全部数据，避免 $N+1$ 问题；要么是引入 $N+1$ 问题，变成钝刀子割肉。**

为了避免这两种情况，很自然地，你会想起这样一种做法：通过查询语句一次性定位所需分页，这样只需要一次查询就能解决问题。不过这么做的困难在于，**分页查询的逻辑要放在哪个对象上，才能保持模型与软件实现的关联。**

一种做法是**为订阅（Subscription）构造一个独立的 Repository 对象**，将逻辑放在里面（也是 Spring 推荐的做法）：

 复制代码

```
1 interface SubscriptionRepository extends ... {  
2     public Page<Subscription> findPaginated(int from, int size);  
3 }
```

这种做法的问题就是会导致**逻辑泄露**。Subscription 被 User 聚合，那么 User 所拥有的 Subscription 的集合逻辑应该被封装在 User 中，这样才能保证 User 是“逻辑丰富的模

型”，因为**非聚合根提供 Repository 是一种坏味道**。

如果聚合到 User 上是可行的吗？其实也不行。因为这么做会将技术实现细节引入领域逻辑中，而无法保持领域逻辑的独立。代码如下：

[复制代码](#)

```
1 public class User {
2     public List<Subscription> getSubscriptions() {
3         ....
4     }
5
6     public List<Subscription> getSubscriptions(int from, int size) {
7         return db.executeQuery(...);
8     }
9 }
```

造成这种两难局面的根源在于，**我们希望在模型中使用集合接口，并借助它封装具体技术实现的细节**。因为在我们的概念中，**内存中的集合与数据库是等价的，都可以通过集合接口封装**。但实际情况是，我们无法忽略数据库带来的额外作用与开销，内存中的集合与数据库并不是等价的，封装失败。

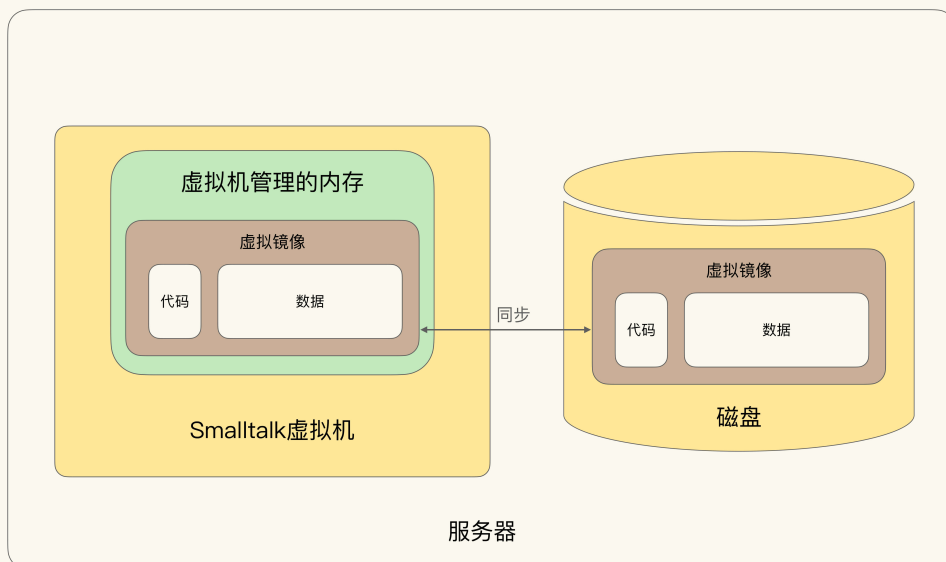
那么，为什么在我们的概念中，我们会认为内存中的集合与数据库是等价的，是可以通过集合接口封装的呢？这就要从面向对象技术的开端——Smalltalk 系统说起了。

Smalltalk 中集合与数据库是等价的

鲜少有人了解的一个事实是，**集合与数据库的等价，在早期面向对象系统 Smalltalk 中是一个基础概念**。

这其实不难理解，如果对象中包含的状态可以映射为数据库表中的一列，那么一组对象自然就可以映射成一张表了。有了一张表，自然也就可以看作是一个最小的数据库了。

Smalltalk 作为面向对象系统内置的一套**面向对象数据库**（Object Oriented Database）。于是在使用 Smalltalk 系统时，集合与数据库是无差别的。如下图所示，展示了 Smalltalk 系统的构成。



如上图所示，Smalltalk 中除了虚拟机（Virtual Machine）之外，还有虚拟镜像（Virtual Image，VI）。**虚拟镜像相当于虚拟机内存的持久化保存。**每次虚拟机启动的时候，都会把虚拟镜像中的信息读入到内存中以恢复状态，而 Java、.NET 虚拟机并不会保留之前内存中的信息。

打个比方。Smalltalk 虚拟机的启动过程，类似于 VMWare 虚拟机上的 Pause，内存中的所有状态都被持久化保存了，那么我们可以继续从之前暂停的地方开始。而其他语言的虚拟机则是彻底地重启（Restart），并不会保留之前的结果。

在 Smalltalk 的虚拟镜像中，存储着所有的代码和数据。我们称呼 Smalltalk 为一个面向对象系统，而不仅仅是面向对象语言。因为它既是一门面向对象语言，也是一个虚拟机的操作系统，还是一个面向对象数据库。

那么在这样的系统中，Smalltalk 中的集合（无论是 Array、List，还是什么）就等同于一个数据库。只要这个集合被创建出来，除非显式地通过垃圾回收销毁，否则它就会被虚拟镜像持久化。**垃圾收集和持久化一样，都可以被看作是对象生命周期（Object Lifecycle）的一个状态。**

因此，数据库中的数据和非数据库中的数据，都可以通过 Collection 表达，我们不用再去区分它。而在 Smalltalk 中，这种等价关系深刻影响了我们的思路和建模习惯，毕竟我们所熟知的面向对象语言都从 Smalltalk 中借鉴了大量的概念。而早期的建模者也大都来自

Smalltalk 社区，比如 Peter Coad、Kent Beck、Martin Fowler 等（当然我也是）。所以我们学习建模方法的时候，或多或少有一些源自 Smalltalk 社区的习惯。

然而随着时代的发展，这种习惯在多层架构（Mulit-tier Architecture）下遇到了挑战。那么接下来我们就看看多层架构是怎么彻底割裂了集合与数据库的。

多层架构彻底割裂了集合与数据库

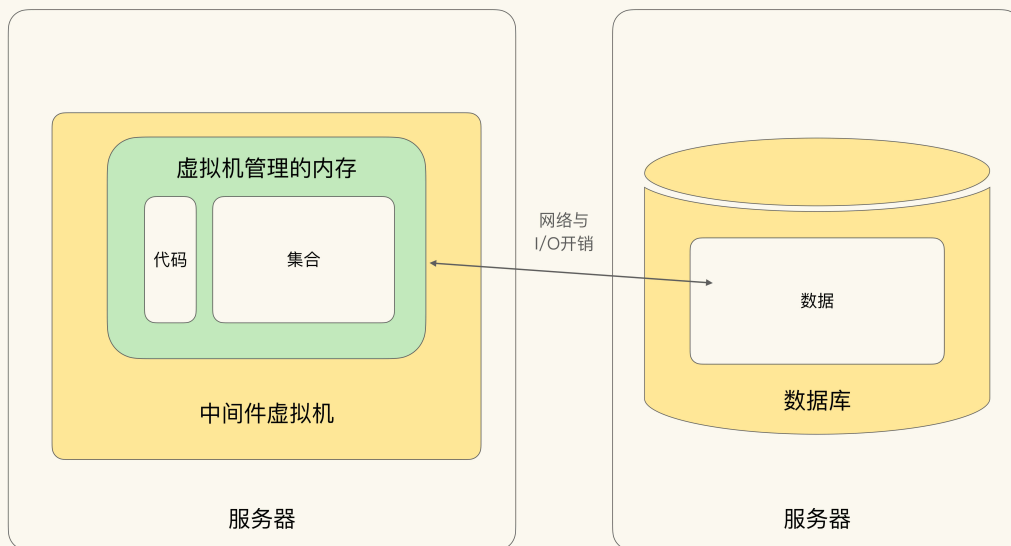
Smalltalk 在中国未曾真正地流行过，大多数人是从 C++ 甚至是 Java、.NET 开始接触面向对象编程的。而无论是 C++、Java，还是后来的面向对象编程语言，其实都不是完整的面向对象系统。

集合作为面向对象中的重要概念，被吸纳入这些语言后，就去掉了与数据库的强关联（毕竟这些语言默认不带有数据库模块）。因此，集合虽然在概念上仍然等同于数据库，但如此简单粗暴的建模，就会遇到我们前面提到的“是否 N+1”的两难选择。

那么 Java、.NET 这些语言为什么要这么做呢？它们为什么不能像 Smalltalk 一样，内涵完整地面向对象数据库呢？这其实是架构风格演化的结果。

我们明显可以发现，**Smalltalk 是典型的单机单体架构**。从操作系统、数据库到应用程序，都在一个虚拟机上，就好像是一台小型机一样。这样的**架构结构无法水平扩展**（horizontal scaling），**只能垂直扩展**（vertical scaling）。这就意味着我们需要**不断提高单点的计算能力，才能提高整个的容量与吞吐**。这并不符合架构的整体发展趋势。

进入 21 世纪后，随着对系统容量要求的增加，易于水平扩展的三层架构逐渐成为行业主流。而数据库作为一个独立组件，则不再与应用合体。示意图如下：



于是，Smalltalk 的面向对象系统在这种架构下分别变成了：应用程序中间件（对应 Smalltalk 虚拟机）和数据库（对应 Smalltalk 虚拟镜像的持久化数据部分）。

多层架构彻底割裂了对象集合与数据库，这对我们实现领域模型建模提出了挑战，对 Collection 逻辑的建模也就难以摆脱具体实现细节了。那就是我们**必须明确哪些是持久化的数据，并对它的一些逻辑区别对待**。这就是原味面向对象范型（Vanilla Object Oriented），在架构风格演化过程中遇到的挑战。

几乎所有实施过 DDD 的人，都在这个问题上挣扎过。如果对于如此常见且如此简单的分页功能，我们都无法将模型与软件实现关联的话。那么，面对更复杂的问题时我们要怎么办呢？这是大多数怀揣理想准备实施 DDD 的人，碰到的第一个阻碍。

关联对象就是一种解决这个问题的设计模式。

关联对象

关联对象，顾名思义，就是**将对象间的关联关系直接建模出来，然后再通过接口与抽象的隔离，把具体技术实现细节封装到接口的实现中**。这样既可以保证概念上的统一，又能够避免技术实现上的限制。

现在让我们再来看看极客时间专栏的例子，如果使用关联对象，如何帮我们避免 N+1 和逻辑泄露的问题。

使用关联对象实现聚合关系

首先我们需要定义关联对象，因为我们需要表达的是 User 与 Subscription 间的一对多的关系，那么最简单的命名方法是将两个对象名字组合，从而得到关联对象的名字 UserSubscriptions：

[复制代码](#)

```
1 public interface UserSubscriptions extends Iterable<Subscription> {
2     List<Subscription> subList(int from, int to); //分页
3     double getTotalSubscriptionFee(); //获取总共花费
4     int count(); //获取总订阅数
5     Iterable<Subscription> sort(...);
6     ....
7 }
```

当然，我们最好是从业务上下文出发，寻找更具有业务含义的名字，毕竟我们是要形成统一语言的，源自业务的名字总是更容易被非技术角色所理解。

比如这里我们要表达的是用户已经订阅的专栏，或者是用户已经购买的专栏，那么 PaidColumn，甚至 MySubscriptions，都不错，看哪一个更容易被团队接受了。我这里选择 MySubscriptions，于是 User 对象就变成了这样：

[复制代码](#)

```
1 public interface MySubscriptions extends Iterable<Subscription> {
2     ...
3 }
4
5
6 public class User {
7     private MySubscriptions mySubscriptions;
8
9     public MySubscriptions getMySubscriptions() {
10         return mySubscriptions
11     }
12 }
```

那么之前对于分页和计算的调用也就相应地变为：

 复制代码

```
1 user.getMySubscriptions().subList(0, 10);
2 user.getMySubscriptions().getTotalSubscriptionFee();
```

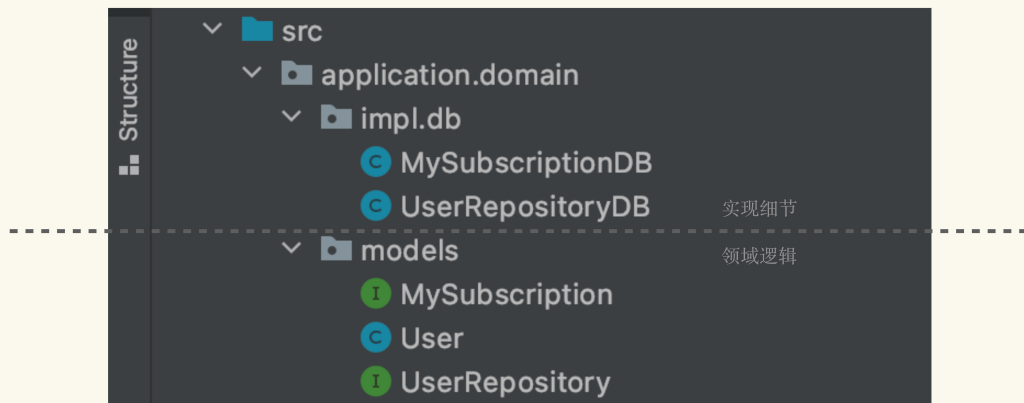
很明显，我们没有逻辑泄露，User 是 Subscription 的聚合根，那么与之相关的逻辑也仍然被封装在 User 的上下文中，当然是进一步被封装在关联对象中。

那么我们怎么解决持久化的问题呢？怎么从领域对象中，移除掉对技术实现的依赖呢？秘诀就在于**接口与实现分离**。代码如下：

 复制代码

```
1 package model.impl.db;
2
3
4 public class MySubscriptionsDB implements MySubscriptions {
5     ...
6     private User user;
7
8
9     public List<Subscription> subList(int from, int to) {
10         return db.executeQuery(...);
11     }
12     ...
}
```

在这里，我们将与数据库访问相关的逻辑毫不避讳地封装到 MySubscriptionsDB 中。不过，作为领域对象的 User 类，并不会知道它使用了数据库，因为它仅仅通过 MySubscriptions 接口，访问它所需要的功能。此时我们可以通过简单的分包的策略：



从上图中包的架构上看，模型的包中有 `MySubscriptions`、`User`、`UserRepository`。与数据库相关的代码的包里有 `MySubscriptionsDB` 和 `UserRepositoryDB`。于是，我们成功地将核心的领域逻辑与实现细节分开了。

当然最后还有一个问题：**如何将 `MySubscriptionsDB` 与 `User` 结合在一起**？最直接的做法就是这样：

[复制代码](#)

```
1 public UserRepositoryDB implements UserRepository {
2     ...
3     public User findBy(long id) {
4         User user = .....;
5         return setMySubscription(user);
6     }
7
8     public List<User> findBy(...) {
9         List<User> user = .....;
10        return user.map(user -> setMySubscription(user));
11    }
12
13    private User setMySubscription(User user) {
14        user.setMySubscriptions(new MySubscriptionDB(db, user));
15        return user;
16    }
17 }
```

因为 `User` 是聚合根，从数据库中读取的 `User` 对象都需要从 `UserRepository` 中获取。那么，在 `UserRepository` 的具体实现中为 `User` 对象设置 `MySubscription` 对象，是再自然

不过的选择了。

当然更简洁漂亮的做法，是通过 Java CDI API 或是框架提供的生命周期实践监听器，来完成关联对象的设置。

隔离技术实现细节与领域逻辑

通过引入关联对象，我们可以更好地隔离领域逻辑与技术实现细节。在这里，我们还是用极客时间专栏的例子来解释。

如果这时候 Subscription 信息并不是存在数据库中，而是通过 RESTful API 从另一个系统中获取的。那么，我们只需提供另一个 MySubscriptions 的实现就可以了：

 复制代码

```
1 package model.impl.api;
2
3
4 public class MySubscriptionsAPI implements MySubscriptions {
5     ...
6     private User user;
7
8
9     public List<Subscription> subList(int from, int to) {
10         return client.findSubscriptions(...);
11     }
12     ...
```

这种改变并不会传递到领域逻辑层，对于分页和计算的调用仍然为：

 复制代码

```
1 user.getMySubscriptions().subList(0, 10);
2 user.getMySubscriptions().getTotalSubscriptionFee();
```

RESTful API 的性能瓶颈和需要调整的地方与数据库不同，这种变化都被关联对象的接口封装隔离了。

从面向对象编程的角度来说，我们很容易理解为什么关联对象可以带来如此多的好处。在诸多面向对象的最佳实践中，有一条是说要尽可能避免使用原始类型（primitive type）。

因为原始类型没有对概念进行提取，也缺乏封装，所以我们应该尽可能地使用自定义的类去替换它们。

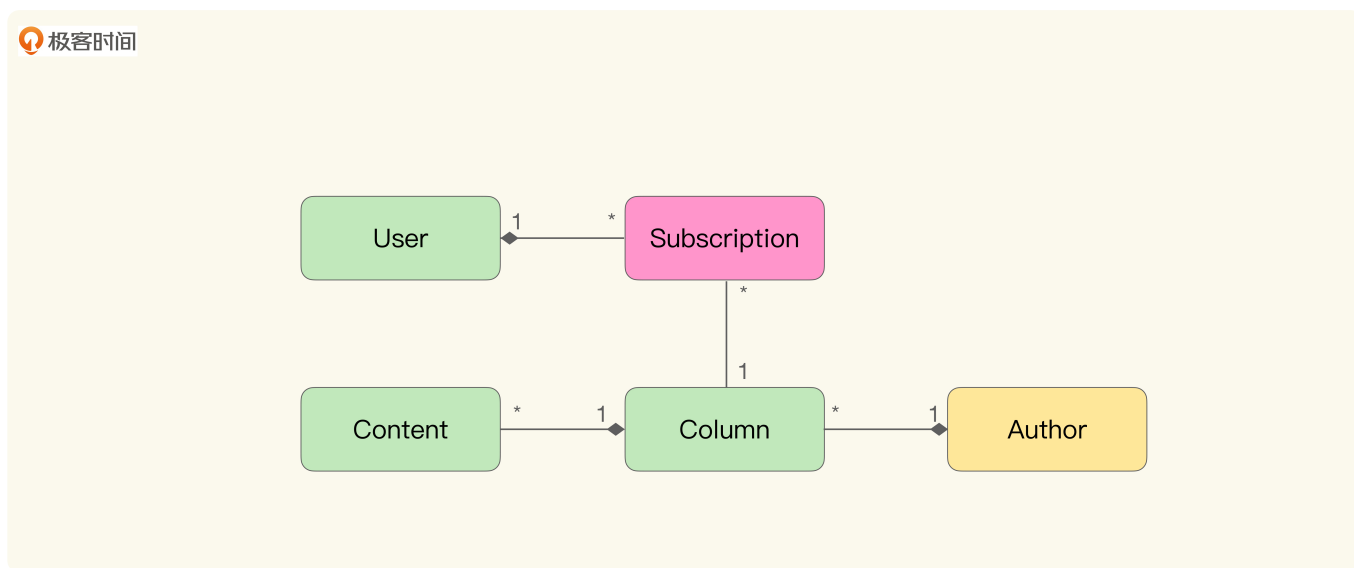
不过如果我们把语言内提供的集合类型（List 等）也当作原始类型的话，关联对象就是对这一条建议自然的扩展：**使用自定义关联对象，而不是集合类型来表示对象间的关联。**

通过集体逻辑揭示意图

关联对象除了可以帮助聚拢逻辑、隔离实现细节之外，还能从概念上帮助我们获得更好的领域模型，因为关联对象是对集体逻辑的直接建模。

所谓**集体逻辑**，是指个体不具备，而成为一个集体之后才具备的能力。哪怕是同一群个体，组成了不同的集体，就会具有不同的逻辑。

我们仍然回到极客时间专栏的例子，来帮助你理解。比如我们现在的模型是这样的：



复制代码

```
1 public class User {
2     private List<Subscription> subscriptions;
3
4     ....
5 }
6
7
8 public class Column {
9     private List<Subscription> subscriptions;
10
11     ....
```

在这段代码中，User 中的 List 表示用户已订阅的所有专栏，而 Column 中的 List，则表示所有订阅了专栏的用户。虽然同为 Subscription 的集合，但是当它们组成集体时，在不同的上下文中则具有不同的含义。

那么如果显式地表达为关联对象，可以进一步澄清我们的意图，得到揭示意图的接口（Intention Revealing Interface）。代码如下：

 复制代码

```
1 public class User {
2
3
4     public static interface MySubscriptions extends Iterable<Subscription> {
5         ...
6     }
7
8     private MySubscriptions mySubscriptions;
9
10    ...
11 }
12
13
14 public class Column {
15
16
17     public static interface MyReaders extends Iterable<Subscription> {
18         ...
19     }
20
21     private MyReaders myReaders ;
22
23
24     ..
```

在这段代码中，我们通过引入关联对象，可以将这两个不同的集体变更为 User.MySubscriptions 和 Column.MyReaders，然后在各自的上下文去定义不同的集体逻辑。

比如我订阅的专栏可以计算我一共付了多少钱，而在我的读者中，可以计算订阅者对专栏的平均打分情况，示意代码如下：

 复制代码

```
1 public static interface MySubscriptions extends Iterable<Subscription> {  
2     double getTotalFee();  
3 }  
4  
5  
6 public static interface MyReaders extends Iterable<Subscription> {  
7     double getAverageRating();  
8 }
```

小结

我们来简单地总结一下。随着架构风格的演化与改变，面向对象技术中的集合，从与数据库完全等价，变成了与数据库完全割裂的东西。

然而在概念上，我们仍留有**通过集合封装内存中的对象与数据库中的数据**的习惯。这使得我们在使用领域驱动设计的时候，特别是使用聚合关系的时候，变得左右为难：**要么放弃性能，获得更好的模型；要么泄露逻辑，以得到可接受的性能。**

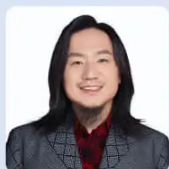
但是关联对象，则可以让我们在更加明确揭示意图的同时，去解决性能与逻辑封装的问题。我个人从很早就开始将关联对象作为实现聚合关系的默认方法了，它从未让我失望（it never fails me）。我建议你也试一试，这样我就不用给你解释什么叫惊喜了。

思考题

关联对象实际上是通过将隐式的概念显式化建模来解决问题的，这是面向对象技术解决问题的通则：永远可以通过引入另一个对象解决问题。那么在领域模型中还有哪些隐式概念呢？这些概念会给我们带来什么麻烦呢？

“

关联对象是一种古老的设计/分析模式，使用它建模领域驱动设计中的聚合与关联关系，可以让我们在更加明确地揭示意图的同时，去解决性能与逻辑封装的问题。



如何落地业务建模

徐昊 | ThoughtWorks 中国区 CTO



识别二维码
免费试读

”

欢迎把你的思考和想法分享在留言区，我会和你交流。同时呢，我也会把其中一些不错的回答置顶，供大家学习。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇

03 | 我们要怎么理解领域驱动设计？

下一篇

05 | 跨越现实的障碍（中）：富含知识还是代码坏味道？

更多学习推荐

Java 面试必考 300 题
最新汇总

限时免费领取

精选留言 (21)

💬 写留言

Geek_3531cc 置顶
2021-07-01

感谢老师的好文章。我这边有个疑问，专栏聚合了内容同理也可以用关联对象模式，那如果作者需要对专栏里的其中一个内容进行修改，需要先获取专栏然后通过关联对象获取要修改的内容进行修改，再将整个专栏聚合进行保存，这样是否还是没有保障到性能呢？

作者回复: 可以在关联对象上直接增加方法，关联对象表示聚合逻辑。灵活很多

💬

👍 3

阿鸡 (醉)
2021-07-02

想请问为什么不直接在UserRepository中添加关于分页的逻辑？虽然能感觉到不太恰当，但是好像也没暴露逻辑，并且user也没依赖具体db实现

展开 ∨

作者回复: 这样user就退化为纯粹的无逻辑数据对象了，也就是贫血对象



webmin

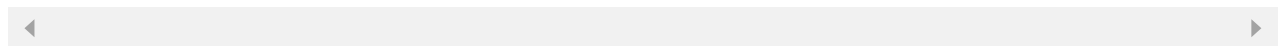
2021-07-03

以前只是觉得语言中对象与关系型数据库之间的转换非常不自然，从未问过为什么会是这样，感谢老师今天帮我理清这层关系，原来二者在单体架构时期是等价，是因为后来分层以后才被割裂了，感觉现在的多层是把以前在语言内部实现的机制给放大化了，且为了通用性把CRUD等细节暴露出来了，导致编程语言在使用时感觉不那么自然和一体化了。

...

展开 ∨

作者回复: 之前在美国的DDD大会上很多实践者严肃地讨论要不要放弃关系型数据库。



Jxin

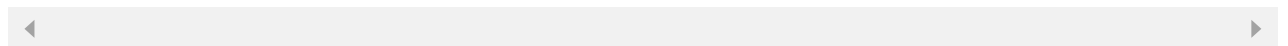
2021-07-02

关于内容：

1.基础类型偏执。这是一个坏味道，但仅在个人或者小团队的小项目中能看到它被认真对待（也只有刻意训练，践行代码健身操时能被认真对待）。一旦项目变大时间变长人数变多，不知不觉就被抛弃了（毕竟这么写对于不理解的人来说真的会被骂死。一段代码要求读者有比较高的认知才能有比较好的可读性。以教育读者为前题的同理心写出来的...

展开 ∨

作者回复: pojo是幻想 cqrs是邪教



Oops!

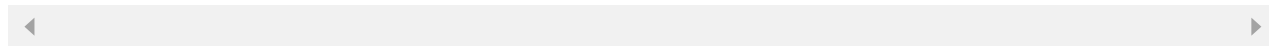
2021-07-01

集合是面向对象模型中广泛存在的概念,如果全都使用关联对象进行建模,是否会导致类爆炸呢? 有什么可以遵循的规则或者方法来鉴别哪些隐式的集合概念使用关联对象进行建模

比较好, 哪些则不然呢? 是不是如果两个对象之间仅仅是简单的包含关系, 可以先用系统提供的集合容器来建模实现, 等到业务复杂了, 需要对这个集合进行除了增删改查之外的操作时, 再使用关联对象进行建模?...

展开 ▾

作者回复: 需要持久化或者来着三方系统的聚合



狩月

2021-07-01

这个模式很接地气啊！期待更多内容

展开 ▾

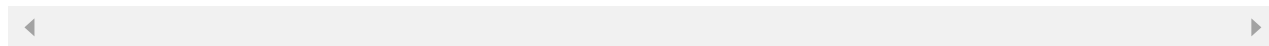


邓志国

2021-07-18

这个user在不同的聚合根里是不是会有多个同名对象，然后实现不一样？否则user要处理的业务会爆炸

作者回复: 看后面



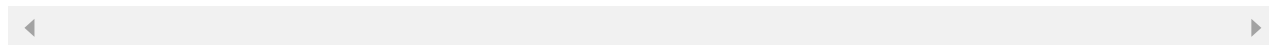
马若飞

2021-07-15

“集合作为面向对象中的重要概念，被吸纳入这些语言后，就去掉了与数据库的强关联”——从语言演进历史得出问题本源，膜拜！

展开 ▾

作者回复: 哥 别这样



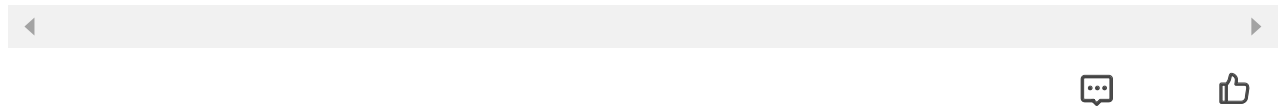
rusess

2021-07-11

感谢老师的好文。我的问题是，如果是要获取用户列表，然后在每行用户信息上面同时显示一个专栏列表，通过MySubscriptions的方式查专栏，这样又会变成 N + 1 查。实际

的DB实现 应该是通过join的方式一次性查询，那我在模型上应该如何定义方法？

作者回复: 不会 my subscription的实现里也是用join



大碗

2021-07-08

请问下老师，用户新订阅一个专栏的写法应该是什么样子的？比如

UserApplication类

```
public void subscribe(User user, Subcription subcrition){  
    user.subscribe(subcrtion);  
    userRepository.save(user);...
```

展开



tongzh

2021-07-05

关键对象模式，不仅成功解耦了领域模型和具体实现，还成功构造出了富含知识的模型



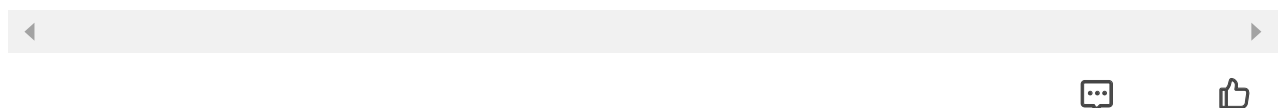
Reason

2021-07-05

getTotalSubscriptionFee应该是一个领域内的计算逻辑吧？通过接口定义的话，计算逻辑是不是就泄露到领域之外了？

展开

作者回复: 关联对象是领域模型一部分 为什么在领域之外？



Geek_5cfbf30

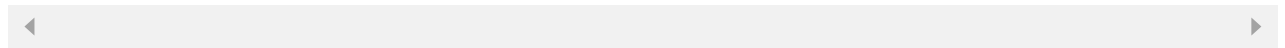
2021-07-05

cqrs 是为了防止复杂的查询增加了领域逻辑的复杂度, 毕竟读的请求逻辑不会很复杂, 会需要获取多个聚合根才能构造出对象, 也是降低落地DDD的复杂度, 毕竟查询还是可以按照以前直接写sql的方式, 而且可以优化性能, 但是边界确实容易扩大. 我们没有再重新定义一个新的读模型, 因为用于承载聚合的数据模型已经够用了.

之前也有疑惑碰到聚合有一对多或者多对多的情况如何处理, 之前是通过拆分成两个聚合...

展开 ▾

作者回复: 是不是关系性数据 并不影响在概念上是不是聚合 聚合也不是关系型数据库的概念



赵晏龙

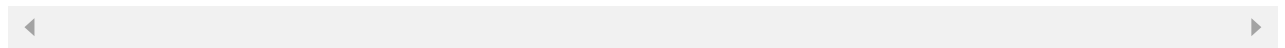
2021-07-05

关联对象我更多的是用来解决业务上的多对多关系，至于分页这个逻辑，我倒是目前还没遇到过这样的场景，不过看完有一些疑问：Subscription虽然通过接口隔离了数据库操作，但是如果Subscription本身有一些业务逻辑呢？是否就考虑把接口换成抽象类，在抽象类中实现业务？

另外，在我看来，这种方式应该只在遇到数据库性能问题的时候使用，不应该作为通用...

展开 ▾

作者回复: 作为通用方法建模的是集合逻辑。有整整五年 我们在全球范围内 看到使用cqrs的项目几乎都失败了 仍然很多人前赴后继 不是邪教是什么



2

1



箭和方糖

2021-07-04

这个模式确实很巧妙，称得上是惊喜。然而它却是情理之中，利用面向对象的接口抽象实现，妙！



四仔

2021-07-04

越看越觉得值！！！！

展开 ▾

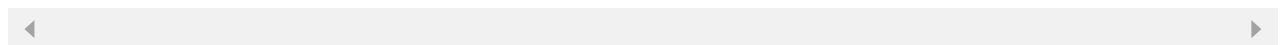


吴鹏

2021-07-02

看到作者thoughtWorks背景毫不犹豫就订阅了，看到这里就感觉值回票价了

编辑回复: 满是惊喜，值得拥有



**大海浮萍**

2021-07-02

我们最近在做聚合落地的时候确实遇到了性能问题，有两个问题想请教一下：

第一个问题，以user-subscription聚合为例，使用关联对象，那么在聚合的持久化上，是不是得分为两步？第一步是先在user实体中使用关联对象接口先持久化subscription的实体，第二步是等整个聚合计算结束后，再持久化聚合根用户，这样算是真正的业务关注点与技术关注点分离吗？毕竟你是在user实体中显示地调用接口做db操作。...

展开 ∨

作者回复: 持久化user是通过user repo进行的，user不会调用任何db 它只调用接口。从实际执行上看 user的确夹杂db行为，但是从user来看并没有，db行为被封装了



1

**威**

2021-07-01

希望老师后面能介绍一些异步回调怎样结合到领域建模里面去

**狩月**

2021-07-01

有点像控制的更精细的ORM Lazyload?

展开 ∨

